

Racket

[1] Vocabulario

➡ Valores

1. Numéricos
2. Booleanos
3. Cadenas
4. Imágenes

➡ Primitivas

1. Palabras clave
2. Operadores matemáticos
3. Operadores sobre cadenas
4. Operadores booleanos

[2] Syntax

➡ Notación prefija

1. Expresiones S

(`name` arg1 arg2 ...)

2. Codificación de caracteres Unicode

(<`operador`> <val1> [val2] [...])
(<`funcion`> <val1> [val2] [...])

[3] Semántica

➡ Tipos de datos

[4] Programas interactivos

➡ Estado

El estado es el conjunto de valores que cambia cuando ocurre un evento. Requiere cuidado al diseñarse para que sea eficiente y cómodo.

1. Estado inicial

Se requiere definir un estado inicial que es el que será presente al iniciar un programa.

2. Función interpretar

Interpreta el estado para mostrar al usuario la información necesaria.

➡ Expresión big-bang

La expresión big-bang se encarga de manejar el estado en un lenguaje funcional como racket. La syntax es:

```
(big-bang INICIAL ;(El estado inicial)
  [to-draw interpretar]      ;manejador de evento obligatorio
  [<evento1> <manejador1>]   ;manejador opcional 1
  [<evento2> <manejador2>]   ;manejador opcional 2
  ...)
```

[5] Estructuras

➡ Definición de estructuras

La definición de una estructura:

```
(define-struct <nombre> [<campo1> ... <campoN>])
```

También define:

1. Un constructor: ; make-nombre: campo1 .. campoN -> nombre
2. Selectores: ; nombre-campoN: nombre -> campoN
3. Un predicado: ; nombre?: Any -> Boolean

➡ Evaluación de estructuras

```
(posn-x (make-posn a b)) ;*** Notar como se enumera la ley ***
== <def. de posn (ley 1)>
a
```

```
(posn-y (make-posn a b)) ;*** Notar como se enumera la ley ***
== <def. de posn (ley 2)>
b
```

[6] Listas

➡ Propiedades

1. Constructores

Son:

1. '(): La lista vacía.
2. cons: Toma un elemento y una lista y devuelve una nueva lista.

2. Selectores

Son:

1. first: Devuelve el primer elemento de la lista.
2. rest: Devuelve la lista sin su primer elemento.

3. Predicados

1. empty?: Verdadero si es una lista vacía. Falso en cualquier otro caso.
2. cons?: Verdadero si es una lista no vacía. Falso en cualquier otro caso.

➡ Ejemplos:

```
'()
(cons "holis" '())
(cons 2 (cons 3 (cons 4 '()))))
```

Formas abreviadas:

```
empty
(list "holis")
(list 2 3 4)
```

[7] Patrones

→ ¿Que son?

Son funciones generalizadas que abstraen justamente patrones de comportamiento de funciones sobre listas. Ayudan a evitar la repetición y redundancia.

➡ **Filter**

El patrón `filter` toma un predicado y una lista L y devuelve una nueva lista L_1 con elementos que pertenecían a L . Justamente filtra L y en L_1 quedan los elementos que hayan «sobrevivido» el filtro. Su *signatura* es la siguiente:

```
; filter: (X -> Boolean) (Listof X) -> Listof X
```

Una posible implementación:

```
(define (filter pred lst)
  (cond [(empty? lst) empty]
        [(pred (first lst)) (cons (first lst) (filter pred (rest lst)))]
        [else (filter pred (rest lst))])
```

➔ Map

Su signatura es la siguiente:

```
; map: (X -> Y) (Listof X) -> (Listof Y)
```

Una posible implementación:

```
(define (map func lst)
  (cond [(empty? lst) empty]
        [else (cons (func (first lst)) (map func (rest lst))))]))
```

➔ Foldr

Su signatura es la siguiente:

```
; foldr: (X Y -> Y) Y (Listof X) -> Y
```

Una posible implementación:

```
(define (foldr func init lst)
  (cond [(empty? lst) init]
        [else (func (first lst) (foldr func init (rest lst)))]))
```

→ Definiciones locales

```
(define (f ..)
  (local
    (<definicion1>
```

([<definicionN>](#))
expr)